CHEAT SHEETS HERO

Nuxt 3 Server-Side Essentials

A concise guide to leveraging Nuxt 3's powerful server-side features, from API routes and middleware to database integration and robust deployment strategies using Nitro.

Nuxt 3 Server-Side Essentials

SERVER ROUTES & HANDLERS

File-based Routing: Create API endpoints by placing files in `server/api/`. Each file exports a default `defineEventHandler()`.

Example: `server/api/hello.ts`

```
export default defineEventHandler((event) => {
  return { message: 'Hello from Nuxt server!' };
});
```

Access at `/api/hello`.

Define Endpoints with `defineEventHandler()`: This helper wraps your server logic, providing access to the `event` object.

Accessing Low-Level Node.js APIs: The `event.node.req` and `event.node.res` objects provide direct access to the underlying Node.js request and response objects.

Example: Custom headers

```
export default defineEventHandler((event) => {
  event.node.res.setHeader('X-Custom-Header', 'Nuxt Is Great');
  return { status: 'OK' };
});
```

Error Handling: Throw `createError` for automatic error responses.

```
throw createError({
  statusCode: 401,
  statusMessage: 'Unauthorized',
  data: { error: 'Invalid credentials' }
});
```

Pro Tip: Use `event.context` to pass data between middleware and handlers. E.g., `event.context.user = authenticatedUser;`

USEFUL UTILITIES

`getQuery(event)`	Parses and returns query parameters from the request URL.
	Example: `/api/data?id=123`
	`const { id } = getQuery(event);`
	`// id is '123'`

`readBody(event)`	Parses and returns the request body. Supports JSON, URL-encoded, etc.
	Example: POST request with `{ "name": "Nuxt" }`
	`const body = await readBody(event);`
	`// body is { name: 'Nuxt' }`

`sendRedirect(event, url, statusCode)`	Redirects the client to a new URL with a specified HTTP status code (default 302). **Example:** `await sendRedirect(event, '/login', 302);`

`useRuntimeConfig()`	Accesses environment variables and runtime configuration defined in `nuxt.config.ts`. Divided into `public` (client-side access) and private (server-only).
	Example:
	`const config = useRuntimeConfig();`
	`console.log(config.apiSecret); // Server-only`
	`console.log(config.public.baseUrl); // Available client-side`

`getHeader(event, name)` `getHeaders(event)`	Retrieves a specific request header or all headers from the incoming request.
	Example:
	`const auth = getHeader(event, 'authorization');`
	`// auth is 'Bearer <token>'`

`setHeaders(event, headers)` `appendHeader(event, name, value)`	Sets or appends response headers for the outgoing response.
	Example:
	`setHeaders(event, { 'Content-Type': 'application/json' });`
	`appendHeader(event, 'Cache-Control', 'no-cache');`

Common Mistake:	Using client-side composables like `useRouter()` or `useFetch()` directly in server routes/middleware. Stick to server-specific utilities like `getQuery()`, `readBody()`, etc.

MIDDLEWARE & AUTH

Defining Middleware: Place files in `server/middleware/`. These run on every request before any server routes or pages are served. They execute in alphabetical order.

Example: `server/middleware/log.ts`

```
export default defineEventHandler((event) => {
  console.log(`[${new Date().toISOString()}] Incoming request: ${event.node.req.url}`);
});
```

Authentication Guards: Use middleware to protect routes by checking authentication status. If unauthorized, throw an error or redirect.

Example: `server/middleware/auth.ts`

```
export default defineEventHandler((event) => {
  const publicRoutes = ['/api/login', '/api/signup'];
  if (!publicRoutes.includes(event.node.req.url || '/') && !event.context.user) {
    throw createError({ statusCode: 401, statusMessage: 'Unauthorized' });
  }
});
```

Using Composables in Middleware: While most client-side composables aren't available, you can define your own server-side composables or use utilities like `useRuntimeConfig()`.

Data Sharing: Middleware can populate `event.context` for subsequent middleware or server handlers.

```
// server/middleware/user-context.ts
export default defineEventHandler(async (event) => {
  const token = getHeader(event, 'authorization')?.split(' ')[1];
  if (token) {
    // Simulate user lookup
    event.context.user = { id: 'some-id', name: 'John Doe' };
  }
});
```

Pro Tip: For specific route protection, consider using a named middleware inside `defineEventHandler` or checking `event.node.req.url` for more granular control.

DATABASE & INTEGRATION

Working with Databases: Nuxt 3's Nitro server engine allows direct interaction with databases like PostgreSQL, MySQL, MongoDB, or ORMs like Prisma.

Example (Prisma Integration):

1. Install Prisma: `npm i prisma @prisma/client`
2. Initialize: `npx prisma init`
3. Define schema in `prisma/schema.prisma`.
4. Generate client: `npx prisma generate`

Using Prisma Client in Server Handlers: Instantiate your Prisma client and use `await` for all database operations.

```
// utils/db.ts (singleton pattern for Prisma)
import { PrismaClient } from '@prisma/client';

let prisma: PrismaClient;

if (process.env.NODE_ENV === 'production') {
  prisma = new PrismaClient();
} else {
  if (!global.prisma) {
    global.prisma = new PrismaClient();
  }
  prisma = global.prisma;
}
export default prisma;
```

```
// server/api/users.ts
import prisma from '../../utils/db'; // Adjust path

export default defineEventHandler(async () => {
  const users = await prisma.user.findMany();
  return users;
});
```

Environment Variables for DB Connections: Securely manage sensitive data like database URLs using `.env` files. Access them via `useRuntimeConfig()`.

Example `.env`:

```
DATABASE_URL="postgresql://user:password@host:port/database"
NUXT_API_SECRET="your_api_secret_key"
```

Access in Nuxt: `useRuntimeConfig().databaseUrl` (if configured in `nuxt.config.ts`).

Async/Await Best Practices: Always `await` asynchronous database calls to prevent race conditions or unhandled promises. Ensure error handling (`try...catch`) around DB operations.

Common Mistake: Not handling database connection pooling or singleton patterns, leading to excessive connections and performance issues in production. Use a global singleton for your Prisma client or similar ORM.

DEPLOYMENT & HOSTING

Running Nuxt in SSR Mode: By default, Nuxt 3 applications are server-side rendered, meaning HTML is generated on the server before being sent to the client. This improves SEO and initial load times.

Nitro Engine: Nuxt 3 uses Nitro, a powerful server engine that abstracts away underlying server platforms, allowing for flexible deployments to various environments.

Deployment Adapters: Nitro provides adapters to optimize your Nuxt app for different hosting platforms. Configure in `nuxt.config.ts` (`nitro.preset`).

Common Adapters:

- `node`: Traditional Node.js server (default)
- `vercel`: Vercel Serverless Functions
- `netlify`: Netlify Functions
- `cloudflare-pages`: Cloudflare Pages (Workers)
- `firebase`: Firebase Functions
- `static`: Generates a fully static site (no server-side rendering/API routes)

Production Build (`nuxi build`): Compiles your Nuxt application into a highly optimized output for production. Creates a `.output` directory.

Local Production Preview (`nuxi preview`): Starts a local server to test the `.output` directory generated by `nuxi build`. Essential for verifying your production bundle before deployment.

Deployment Flow (General):

1. Ensure `nitro.preset` is set in `nuxt.config.ts`.
2. Run `npm run build` (or `nuxi build`).
3. Deploy the contents of the `.output` directory to your chosen platform.

Pro Tip: Always test your production build locally with `nuxi preview` before deploying to a live environment. This helps catch configuration errors or issues specific to the compiled output.

Common Mistake: Forgetting to configure `nitro.preset` for serverless platforms, which can lead to larger deployment sizes or incorrect function behavior.